An Improved Method for Register File Verification

by

Tong Quan, B.S.

Report

Presented to the Faculty of the Graduate School

Of the University of Texas at Austin

In Partial Fulfillment

Of the Requirements

For the Degree of

Master of Science in Engineering

The University of Texas at Austin

August 2009

The Report Committee for Tong Quan

Certifies that this is the approved version of the following report

An Improved Method for Register File Verification

APPROVED BY

SUPERVISING COMMITTEE:

Supervisor: _____

David Z. Pan

_____

Nur Touba

Acknowledgements

The author would like to thank Professor David Pan for providing his expertise and guidance and Professor Nur Touba for giving his valuable feedback throughout this masters report process. In addition the author would like to thank his various managers and co-workers at IBM who provided much needed support and inspiration throughout his master's school process.

An Improved Method for Register File Verification

By

Tong Quan, M.S.E

The University of Texas at Austin, 2009

SUPERVISOR: David Z. Pan

Register file logic verification historically involves comparing two human generated logic sources such as a VHDL code file and a circuit schematic for logic equivalence. This method is valid for most cases, however it does not account for instances when both logic sources are equivalent but incorrect. This report proposes a method to eliminate this problem by testing logic coherency of various sources with a golden logic source. This golden logic source will be generated by a register file simulation program which has been developed to simulate accurate regfile I/O port data. Implementation of this simulation program for logic verification will eliminate the accuracy problem stated above, in addition the logic simulation time for the new method has also been reduced by 36% compared to the former method.

Table of Contents

List of Figures

CHAPTER 1

Introduction

1.1 Design Space

This focus of this project is in the area of register file logic verification methods and accuracy. We will be looking at some inherent deficiencies in today's logic simulation techniques, and how these deficiencies can lead to logic bugs in design and also possible chip yield loss and field exposure. After exploring some possible alternatives, the goal of this report is to propose a corrective solution to this logic sim problem. This logic simulation solution will improve several areas of the regfile EDA process, namely it will offer improved simulation accuracy and also reduced verification time.

1.2 Problem Definition

The current regfile EDA process typically uses non-golden logic sources as their bases for logic verification which can lead to logic bugs in the regfile. For example, usually during logic sim of a regfile circuit schematic or layout, designers will use VHDL or Verilog source code to test for logic equivalence. The VHDL and Verilog source code themselves are usually generated by the designers and are not systematically verified by a design tool, which means they could have logic problems that have gone unnoticed. These source files usually are only approved during design reviews which are not

foolproof, and also verified during chip simulation which will only cover a small percentage of register file states due to the high number of test cases required. This non-golden verification source used during logic verification can lead to logic bugs in the register file, in addition it can also propagate to chip build if the chip simulation fails to catch the error.

1.3 Problem Importance

As mentioned previously, using a non-golden logic source for verification can lead to logic bugs in the regfile circuit. Logic problems are hard to isolate on a chip level even when they are discovered which leads to unnecessary amount of resources devoted to solving the problem. In addition, if the logic defect goes undiscovered during chip build as it generally does for smaller and less used registers, the design will eventually RIT and be shipped to the field. Logic defects are different than other defects such as parametric or process errors where only a certain percentage of chips are affected. If there is no workaround to the logic bug in the field, the entire batch of chips built with the defect will have to be recalled or replaced which leads to vastly wasted resources and customer depreciation.

1.4 Problem Solution

This report proposes a method to generate a golden logic source for regfile verification. This stimulus / expects verification software solution described in this report will be able to generate cycle accurate I/O port stimulus and expected value output

for a variety of different register files. This golden logic output can then be inputted to different verification tools such as Cadence Ultrasim or IBM MESA to determine logic coherency for regfile circuit schematics, layouts and also VHDL and verilog source code files. There are many benefits to this verification approach. In addition to the more reliable logic verification, this simulation software will also be able to generate various simulation patterns faster than using conventional source code or schematics. These advantages should lead to reduced design time and improved chip reliability in the regfile design process.

1.5 Report Structure

In this report first we are going to take a look at the initial design process and how the current architecture was chosen as the problem solution. Next we will go into some background information on the types of register files that are supported in this new verification approach including the regfile functions and clocking. Next we will look in some detail at how the simulation software goes about generating the regfile logic and the possible test patterns that can be used with the software. The simulation results with explanations of the logic I/O outputs will also be included, and lastly we will talk about some future improvements to this test procedure and also a conclusion.

CHAPTER 2

Background and Approach

2.1 Design Planning

During our initial design phase, we understood that with the array of verification tools at our disposal, all we needed for a problem solution is to provide a single golden logic source from which we can use to verify all the other logic in the design space. This gave us a few options as to which is the logic source we are going to choose to verify. Specifically the logic sources that we were familiar with were VHDL source code files and Cadence circuit schematic files.

Initially we considered developing a source specific simulator for the VHDL source code, however this required automated compile and simulation of the VHDL which we weren't familiar with. Developing a simulator for schematic file was even more difficult. So for practical purposes among other reasons we decided to design a standalone regfile simulator which had the ability to output a set of golden input and output pin combinational states. When used in conjunction with other logic verification suites, this golden I/O pin file can then be used to logically verify both the VHDL source code and the circuit schematic. This seemed to be the best method and we decided to use it as it had the largest number of supported verification sources which gave the user the most flexibility. Using this method will also allow more designers to take advantage of this verification approach which will offer the most benefit.

After deciding on the objective method, next we needed to decide on a coding style for implementation. The choices we had were between using the familiar Perl programming language or a VGEN language which was a high level programming language geared specifically for stimulus pattern generation. The VGEN coding style had an advantage in that it is very simple for specifying input and output pins and for sequentially specifying combinational circuits with clocking. Despite the advantages with VGEN, we decided finally to use Perl for coding mainly due to the easier code transferability as more people were familiar with Perl coding techniques. In addition, the disadvantages with Perl programming can be easily compensated with additional support code.

2.2 Register File Structure

After deciding on the design solution method, our next step was to develop a coding priority for the type of register files that the simulation program is going to support. Since this simulation program will be initially adapted for use within IBM's server team circuit design division, we decided to adapt the program toward those types of register files first. These register files are built from sram array cells with multiple read and write ports. The structure for this type of register is fairly standard with a series of wordline address decoders driving a row of memory cells (6T cells for a 1r1w regfile), and the output going to a precharged bitline which finally gets latched before being outputted to the data_out ports. There are various alternatives to the standard regfile structure such as addition of Abist and Cam ports, however the standard memory cell is

prioritized and will be supported in the logic simulator. The primarily focuses on the coding priority will be on generating the stimulus data for the primary address and data inputs, and the final verification data output. An example of this type of supported register file for simulation is in the figure below:

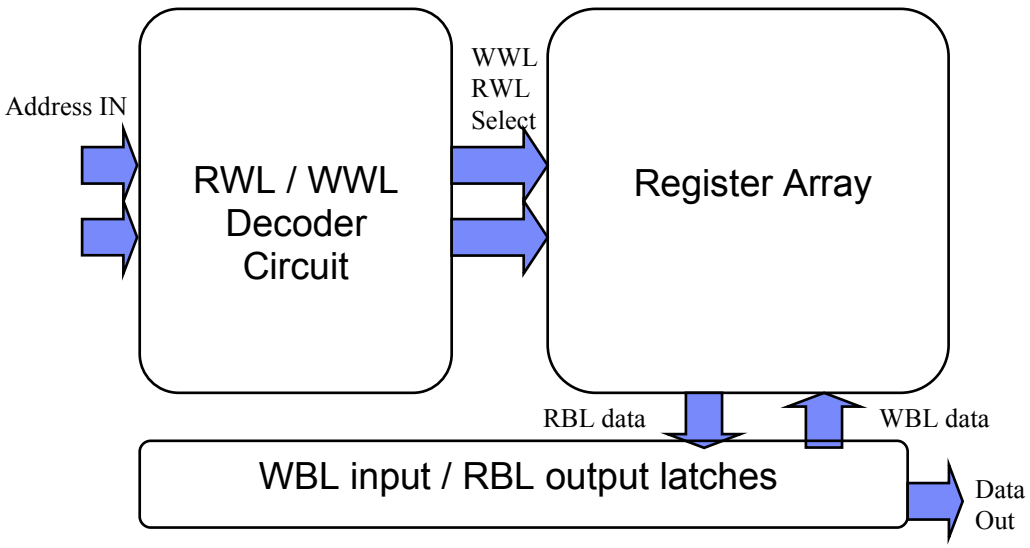

Figure 1: Register file block diagram

2.3 Register File Clocking

The clocking method for this type of register file is full cycle clocking which uses one cycle for each read and write operation. The data for each clocking cycle has to be available during the entire cycle time when the read or write operation is processing. Since the simulation program is entirely logic based, it will not offer simulation results for any delay elements or timing events, although the simulation data can be used to test for such fails in the real circuit. After the read operation, the data first gets latched

internal to the register file before being sent out to the data output pins. This leads to a single cycle delay after the read operation before read data is available at the output. The following figure illustrates the clocking for this supported register file.

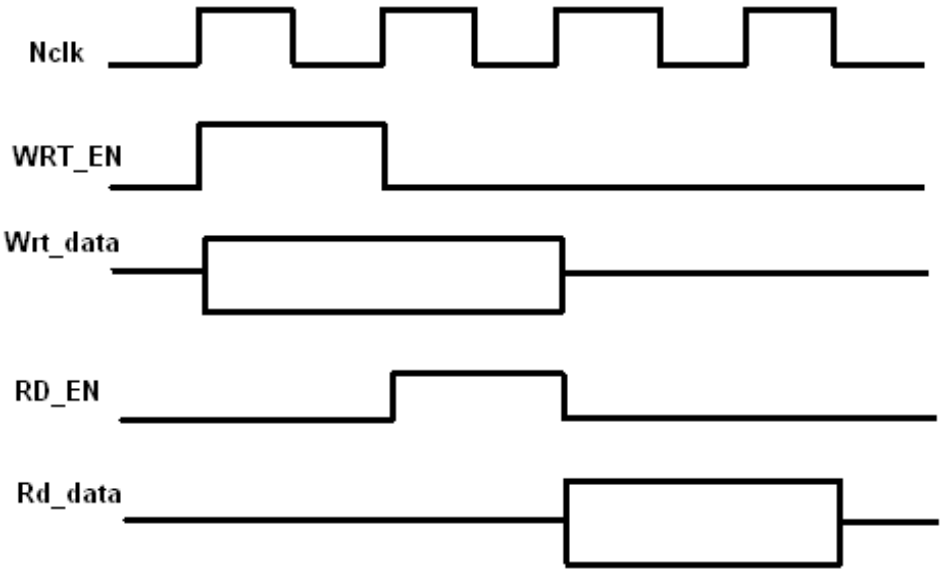

Figure 2: Register file clocking waveform

CHAPTER 3

Design Implementation

3.1 Simulation Procedure

The regfile I/O simulation solution was coded in Perl programming language. The software design flow goes through several stages before the correct register file verification data can be generated. Figure 3 shows a brief outline of these different stages along with the various data flow paths. The section below will take a brief look at each of these blocks in the simulation program.

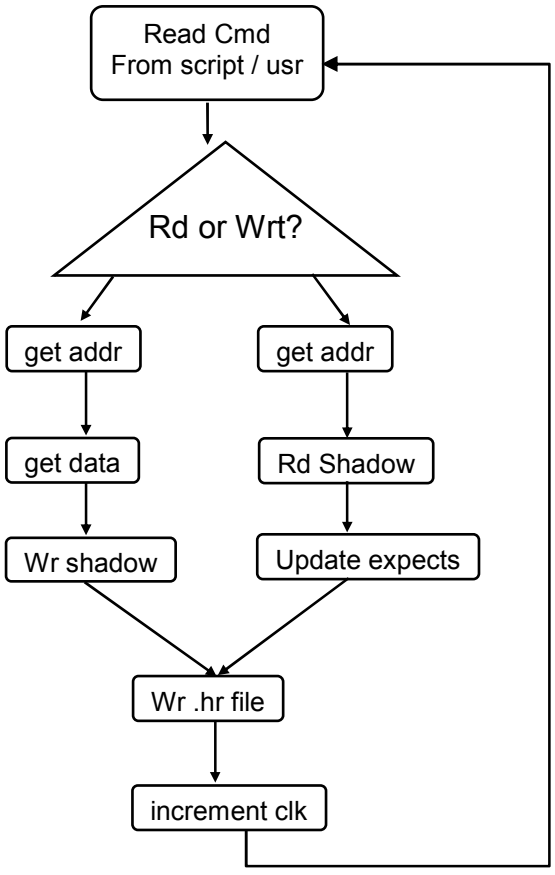

Figure 3: Stims / Expects program flow

When the program first executes it runs through a routine to gather information about the regfile macro under simulation as well as some additional information on the simulation requirements. The regfile macro information is mostly extracted directly from the VHDL header. This information is used to define the size and function specific to the macro. Below are the requirements for those parameters which also defines the size and function of the supported macro in the simulation program

Read Ports – The number of read ports in the macro. Due to the design constraints of array cells, multiple ports in the array tend to have different polarity outputs. This polarity mismatch is not linear but hard coded, thus limiting the current number of read ports to maximum of 6. Figure 4 below shows the different polarities of the two read ports in a read 2 write memory array.

Write Ports – The number of write ports in the macro. There is no logical polarity issue here unlike the read ports. The number of write ports can range from 0 to 9.

Entries – The number of entries or wordlines in the macro. The random address generation algorithm also depends on this number of entries.

Write Data Latch – This specifies whether the write data latch is internal to the regfile macro. Since clocking for the write L2 latch is a full cycle, the clocking for the write data needs to be delayed 1 cycle in this case. This is done by manually adjusting the various outputs after simulation is complete.

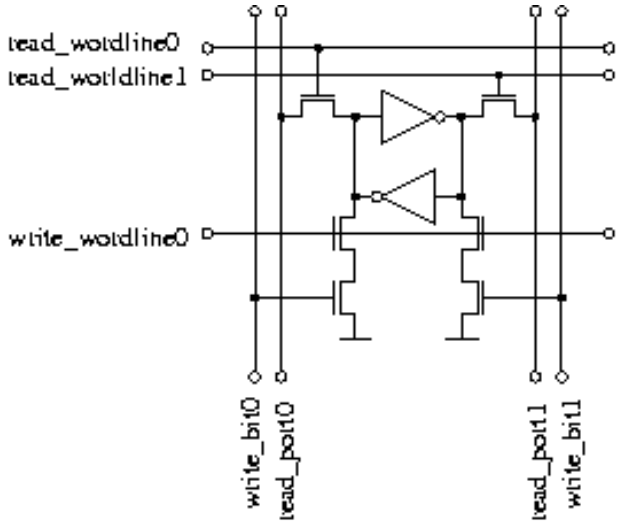

Figure 4. 2read 2write memory array. Source: See References 9

During the actual execution of the read and write command, the simulation program uses a variety of memory hashes and log files to keep track of the internal memory locations. After calling the write_cmd subroutine, the actual write simulation process is relatively simple and involves writing the data and address to a human readable file and also to a shadow array that contains the last address and data written. This shadow array only contains the latest data written to the array and does not carry any change information which keeps the memory usage relatively low.

After the write operation is complete, the user is able to select the read operation to access the first or last address, or the last address written. Before the first read operation, the read data out pins should be at a floating state. This is represented by an "X" in the output files. After the initial read operation, the read data pins now retain the precharge value 0 for non-read cycles and the array cell value for all read cycles. The actual operation for the read cycle starts with the read_cmd capturing and checking the

read operation flags, followed by immediately updating the output file with corresponding read address and enable bits that are inputted by the user.

After the updating the user input data, the simulation program uses the wrExpects routine to update the read output pins and also the shadow array data. Before writing to the read data_out pins, there's also a clock delay applied. This is due to having the data nand2 latch being inside the regfile for all the simulation macros, which leads to read data output being 1 clock cycle behind the actual read operation. Because of this data output latch, in the end of every simulation run there need to be two no op commands to make sure the final read data settles after the last read operation.

3.2 Possible Test Patterns

In the previous section we went into a brief description of the write and read commands as executed by the regfile simulation program. In this section we are going to take a look at some supported test patterns and how users can develop their own targeted tests by combining these read and write commands. In this current version of the simulation software, due to coding priorities only a quicktest (option q) simulation is supported. This quicktest simulation has three separate test patterns that it will try to run on the given macro. The first test consists of targeted read/write operations that simulate certain high failure rate operations. Those operations include reading and writing a high and a low value to the beginning and ending addresses, as well as to the highest and lowest port domains of the macro. These operations are chosen from a logic and circuit design point of view to have the highest probability for failure.

The second and third tests are similar in that they both randomly choose addresses and ports to write to and read from. The sequence for Test 2 issues several random write operations before reading them back. Test 3 also issues random write operations but it reads them back immediately during simulation. Both of these tests when used for hardware verification will test for hardware related defects although for different failure types. Test 2 will primarily focus on leakage fail defects, whereas Test 3 will focus primarily on timing related defects with write and read operations back to back.

Although the supported tests in this version of the verification software do not have an exhaustive test of all the address and ports in the macro, if needed the user does have the ability to explicitly specify the test port and test data of the macro directly in the script. To do this we need to change the read and write command patterns in the "create_test" subroutine of the simulation program. The user is able to change the test patterns specified in this section to specifically target certain areas of the macro. The syntax for the write and read command subroutines are documented below:

write_cmd("r","r","r")

- The write command has 3 input parameters, the port, the address and the data.
- The port parameter can be <r: random> or <n: explicit>.
- The address parameter can be <f: first>, <l: last>, or <r:random
- The data parameter can be <l: low>, <h: high>, <r: random>.

read_cmd("r","p")

- The read command has 2 input parameters, the port and the address.

- The port parameter can be <r: random> or <n: explicit>.
- The address parameter can be <f: first>, <l: last>, or <p: pop last address written>.

Using combinations of these read and write command parameters, the user is able to develop various targeted testers for a regfile macro. For example, for a 4 read 4 write, 10 entry 10 bit register file macro, if the user wanted to do a comprehensive test of the cell port #4 he would use a forloop to randomly test the addresses for the macro. If the length of the forloop is large enough the user is able to obtain large test address coverage even though it is not exhaustive.

Algorithm 1: Random Testing

```
for (counter = 0, counter < 20, counter++)
        write_cmd("4", "r", "h");
        write_cmd("4", "r", "l");
        read_cmd("4", "p");
        read_cmd("4", "p");
end
```

Since this macro is 10 entries, a 20 random address write and read test will run through most of the addresses in that macro, and will likely to catch many logic or circuit problems when used in verification. Anther test scenario would be using the simulator's address targeting abilities to test for early mode and late mode fails. In the example below the last and first addresses are targeted for testing, which corresponds to the longest data path and shortest data path from the address decoder to the memory array.

This when tested against the circuit or layout will verify any late mode and early mode timing path fails.

Algorithm 2: Late and Early Mode Testing

```
for (counter = 0, counter < 20, counter++)
        write_cmd("4", "l", "r");
        read_cmd("4", "p");
        write_cmd("4", "f", "r");
        read_cmd("4", "p");
end
```

CHAPTER 4

Simulation Results

4.1 Simulation Outputs

After the regfile simulation program completes generation of I/O pin values, the final step in the verification process is to associate the resulting I/O output files to the Cadence Ultrasim and IBM MESA verification tools. As shown in figure 5, at the end of simulation, three different output files are generated for debug and further verification purposes. These three output files are the human readable .hr file, the .vec file and the .bat file.

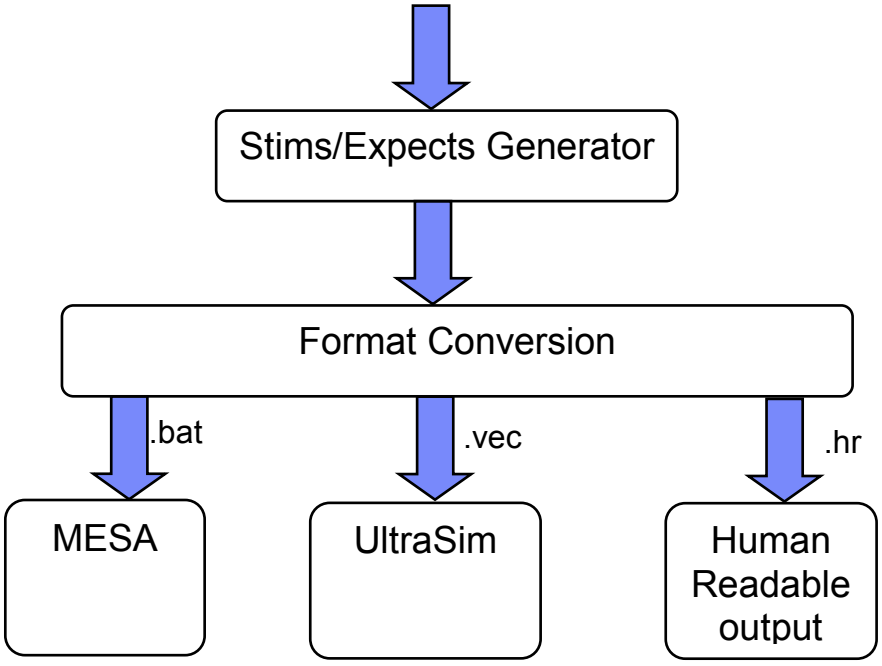

Figure 5: Output file generation flow

4.1.1 Human Readable Output

Of particular interest for failure debug for the user is the .hr or the human readable file which lists out all the I/O pins of the macro in a easy to read clock cycle format. To illustrate, Appendix A1 has an excerpt from the .hr simulation result of a simple 1read 1write 32 entry 11bit regfile. The simulation test in this example is also simple and involves two read commands followed by two commands to read them back. This output excerpt shows the two write addresses being the last bit (bit32) and the first bit (bit0), and the write data high for both write operations. After the write operation the read operation immediately follows with two pop commands which read out the latest address written. The first read address this time is the last write address (bit0) and the subsequent read address is the first address written to (bit32) which is all 1's in this case. Notice there's a latch delay between read operation and the actual r_data_out_0 pin, which is accommodating for the nand2 data capture latch that's internal to the regfile array.

4.1.2 Cadence Ultrasim Output

After the human readable file is generated, next the program will continue to generate the UltraSim vector file for circuit verification. UltraSim in this case is used as an analog to digital hierarchical simulator. It uses fast simulation algorithms with similar accuracy to Spice simulation to generate logic and timing algorithms from a circuit schematic that can be compared to the logical vector file generated from the simulation program.

The information contained in the vector file for UltraSim is similar to the information in the human readable file with a few additional UltraSim specific changes. In the first and second lines in the vector file UltraSim requires the Radix and IO definitions for the primary IO simulation ports. The radix defines how many bits are in that particular signal, and the IO determines whether the port is a primary input or output. Next in the Vname section UltraSim requires all the signal names to be listed in a vertical fashion. Immediately following that there are a few parameters required by UltraSim that's relevant to analog circuit simulation. Those parameters are standard for any analog simulation software, but a brief description of each is provided below:

Table 1: Ultrasim Simulation Parameters

Parameter Name	Parameter Description
tunit	Time unit used for the subsequent variables
trise	Rising slew for the input signals
tfall	Falling slew for the input signals
vih	Logical high voltage value for input signals
vil	Logical low voltage value for input signals
vol	Logical low voltage value for output signals
voh	Logical high voltage value for output signals
period	Circuit lat to lat period

An excerpt from the Cadence Ultrasim file is given in Appendix A2.

4.1.3 IBM MESA Output

In addition to UltraSim simulation, lastly the user has the option to output the IBM MESA batch file. The MESA verification software is different from UltraSim in

that it concentrates primarily on logical VHDL verification instead on circuit or layout verification. This is very useful process since for circuit designers VHDL to circuit schematic verification is often considered to be the golden standard for logic verification, however the VHDL logic coherency check is mostly ignored. Using MESA software to directly verify the VHDL logic for regfiles would account for this lack of test coverage. The batch file used for MESA simulation once again has similar data except here the data is structured to be spread out per cycle. The reason for this is MESA is a cycle reproducible simulator and is able to narrow a logic fail down to the exact cycle of failure. After the data for a particular half cycle is written to the batch file, a single "clock 1" command is issued to differentiate the data for the next half cycle. An excerpt for the 1st half cycle from the MESA batch file for the 32x11 macro is included in Appendix A3.

4.2 Test Time Reduction

In addition to assuring correct logic verification, using the verification method specified in this report was also able to reduce the verification time for a circuit schematic. This is an expected result for this simulation methodology since the golden I/O values are generated by a fast simulation algorithm instead of being simulated from a VHDL or Verilog source code file. For a large 4read 2write 144x74 macro, the total I/O port simulation time was under 7seconds. After incorporating the output file to Ultrasim to verify logic against the circuit schematic, the logic verification time for the entire macro is now 66min. This compared to a similar but less accurate logic verification

method by using Cadence Verity to simulate both the regfile VHDL and the circuit schematic, the total simulation time there was 104min. The simulation time improvement here is 36.5% which is very significant especially when batch testing multiple macros.

Table 2: Simulation time comparison for 4r2w144x74 regfile macro

Simulation Method	**Total simulation time**
Ultrasim simulation with vector file:	66min
Cadence Verity simulation with VHDL:	104min

CHAPTER 5

Future Works and Conclusion

5.1 Future Improvements

The current version of the stims/expects software has many useful features of a logic verification tool. With enough test cycles the program is able to generate stimulus patterns to cover most cells in the regfile array. However due to coding priorities with the current release of the stims/expects simulator it did not include all of the proposed program features and functionalities. Some important features such as exhaustive testing of the entire array, and the ability to target individual address locations were left out in this initial release. However ongoing improvements are being made to the software to add more functionality and robustness to the code. One high priority item currently is to add more flexibility to the user interface for testers to specify their individual test patterns.

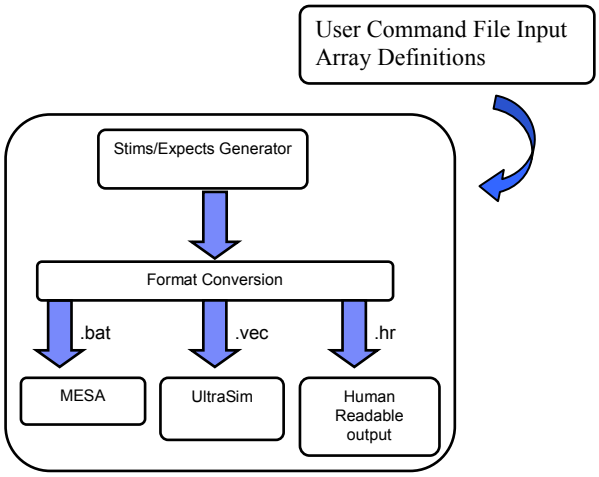

Figure 6: External user command file

As shown in Figure 6, once the user specified test patterns is external to the stims/expects program, it will make the simulation program more stable since it will keep any user modifications outside of the main program code. This user command file will include a simplified method for specifying random or specific data or address, and easier methods to combine write and read commands. In addition to the external user command file, below is a list of other work items currently planned for the stims/expects simulator.

- Exhaustive random test generation for all array addresses
- Ability to specify an address range for test
- Ability to specify individual test addresses for targeted analysis
- Half cycle clocking support
- Outside data latch support
- Register scan port support
- Code optimization
- Abist support
- CAM support

5.2 Conclusion

The regfile logic verification method described in this report provides an easy but necessary way to simulate and verify regfile operations. Due to the lack of industry standard software, the simulator described here becomes a necessary logic verification tool for both hardware description code and hardware schematic entry for correct logic comparison. In addition, by having another circuit verification method, the circuit

designer now is no longer reliant on obtaining the hardware description language before starting schematic entry or verification. The current version of the simulation software can only simulate and output verification files for simple register files, in a way it is only a proof of concept. In the future as more features are added and more regfiles patterns are covered, it should ultimately become another integral part to the chip design process for the EDA community.

APPENDIX

A1: Simulation Test and Human Readable File Excerpt

```
write_cmd("0","l","h");
write_cmd("0","f","h");
read_cmd("0","p");
read_cmd("0","p");
```

```
nest_nclk          1010101010101010
vdn                1111111111111111
gnd                0000000000000000
r_late_en_0        0000000011110000
r_early_en_0       0000000011110000
r_addr_in_0(0)     0000000000110000
r_addr_in_0(1)     0000000000110000
r_addr_in_0(2)     0000000000110000
r_addr_in_0(3)     0000000000110000
r_addr_in_0(4)     0000000000110000
w_late_en_0        0000111100000000
w_early_en_0       0000111100000000
w_addr_in_0(0)     0000110000000000
w_addr_in_0(1)     0000110000000000
w_addr_in_0(2)     0000110000000000
w_addr_in_0(3)     0000110000000000
w_addr_in_0(4)     0000110000000000
w_data_in_0(0)     0000111100000000
w_data_in_0(1)     0000111100000000
w_data_in_0(2)     0000111100000000
w_data_in_0(3)     0000111100000000
w_data_in_0(4)     0000111100000000
w_data_in_0(5)     0000111100000000
w_data_in_0(6)     0000111100000000
w_data_in_0(7)     0000111100000000
w_data_in_0(8)     0000111100000000
w_data_in_0(9)     0000111100000000
w_data_in_0(10)    0000111100000000
r_data_out_0(0)    xxxxxxxxxx111111
r_data_out_0(1)    xxxxxxxxxx111111
r_data_out_0(2)    xxxxxxxxxx111111
r_data_out_0(3)    xxxxxxxxxx111111
r_data_out_0(4)    xxxxxxxxxx111111
r_data_out_0(5)    xxxxxxxxxx111111
r_data_out_0(6)    xxxxxxxxxx111111
r_data_out_0(7)    xxxxxxxxxx111111
r_data_out_0(8)    xxxxxxxxxx111111
```

r_data_out_0(9) xxxxxxxxxx111111
r_data_out_0(10) xxxxxxxxxx111111

A2: Cadence Ultrasim Vector File Output

RADIX 11111111111111111111111111111111111111111
IO iiiiiiiiiiiiiiiiiiiiiiiiiiiioooooooooooooo

VNAME
+ nest_nclk
+ r_late_en_0
+ r_early_en_0
+ r_addr_in_0<0>
+ r_addr_in_0<1>
+ r_addr_in_0<2>
+ r_addr_in_0<3>
+ r_addr_in_0<4>
+ w_late_en_0
+ w_early_en_0
+ w_addr_in_0<0>
+ w_addr_in_0<1>
+ w_addr_in_0<2>
+ w_addr_in_0<3>
+ w_addr_in_0<4>
+ w_data_in_0<0>
+ w_data_in_0<1>
+ w_data_in_0<2>
+ w_data_in_0<3>
+ w_data_in_0<4>
+ w_data_in_0<5>
+ w_data_in_0<6>
+ w_data_in_0<7>
+ w_data_in_0<8>
+ w_data_in_0<9>
+ w_data_in_0<10>
+ r_data_out_0<0>
+ r_data_out_0<1>
+ r_data_out_0<2>
+ r_data_out_0<3>
+ r_data_out_0<4>
+ r_data_out_0<5>
+ r_data_out_0<6>
+ r_data_out_0<7>
+ r_data_out_0<8>
+ r_data_out_0<9>
+ r_data_out_0<10>

tunit ns
trise 0.03
tfall 0.03

```
vih    0.9
vil    0.0
vol    0.2
voh    0.7
period 0.125


   100000000000000000000000xxxxxxxxxxx
   000000000000000000000000xxxxxxxxxxx
   100000000000000000000000xxxxxxxxxxx
   000000000000000000000000xxxxxxxxxxx
   100000001111111111111111xxxxxxxxxxx
   000000001111111111111111xxxxxxxxxxx
   100000001100000111111111xxxxxxxxxxx
   000000001100000111111111xxxxxxxxxxx
   111000000000000000000000xxxxxxxxxxx
   011000000000000000000000xxxxxxxxxxx
   111111110000000000000000011111111111
   011111110000000000000000011111111111
   100000000000000000000000011111111111
   000000000000000000000000011111111111
   100000000000000000000000011111111111
   000000000000000000000000011111111111
```

A3: IBM Mesa Simulation File Output

```
// ** Cycle = : 0
// ** .sim Marker = : [@0]
ALTER nest_nclk '1'b
ALTER vdn '1'b
ALTER gnd '0'b
ALTER r_late_en_0 '0'b
ALTER r_early_en_0 '0'b
ALTER r_addr_in_0(0) '0'b
ALTER r_addr_in_0(1) '0'b
ALTER r_addr_in_0(2) '0'b
ALTER r_addr_in_0(3) '0'b
ALTER r_addr_in_0(4) '0'b
ALTER w_late_en_0 '0'b
ALTER w_early_en_0 '0'b
ALTER w_addr_in_0(0) '0'b
ALTER w_addr_in_0(1) '0'b
ALTER w_addr_in_0(2) '0'b
ALTER w_addr_in_0(3) '0'b
ALTER w_addr_in_0(4) '0'b
ALTER w_data_in_0(0) '0'b
ALTER w_data_in_0(1) '0'b
ALTER w_data_in_0(2) '0'b
ALTER w_data_in_0(3) '0'b
```

```
ALTER w_data_in_0(4) '0'b
ALTER w_data_in_0(5) '0'b
ALTER w_data_in_0(6) '0'b
ALTER w_data_in_0(7) '0'b
ALTER w_data_in_0(8) '0'b
ALTER w_data_in_0(9) '0'b
ALTER w_data_in_0(10) '0'b
clock 1
```

References:

1. Stephen C. Bergman, Robert Shadowen, Zoltan Hidvegi and Matyas Sustik, *Mesa / Mvlsim Simulation System,* IBM Server and Technology Group, February 2006

2. *Virtuoso UltraSim Simulator User Guide,* 2655 Seely Avenue, San Jose, CA 95134: Cadence Design Systems Inc., 2009

3. *Vgen User's Manual Release 3.7,* 3450 Palmer Drive, Cameron Park, CA 95682: Source III, Inc. 2008

4. Sriram Srinivasan, *Advanced Perl Programming,* O'Reilly, First Edition August 1997

5. Brian D. Foy, *Mastering Perl,* 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc. 2007

6. Rama Sangireddy and Arun K. Somani, *Exploiting Quiescent States in Register Lifetime,* Proceedings of the IEEE International Conference on Computer Design 2004

7. Dean L. Lewis and Hsien-Hsin S. Lee, *Testing Circuit-Partitioned 3D IC Designs,* IEEE Computer Society Annual Symposium on Emerging VLSI technologies and Architectures, 2009

8. Prashant Dubey, Akhil Garg and Sravan Kumar Bhaskarani, *Built in Defect Prognosis for Embedded Memories*, IEEE Design and Diagnostics of Electronic Circuits and Systems, 2007

9. Wikipedia Foundation, Inc., *http://en.wikipedia.org/wiki/File:Regfile_cell.png*, Register File article page, 2005

VITA

Tong Quan completed his undergraduate degree at Purdue University with a Bachelor of Science in Electrical and Computer Engineering in 2005. From fall of 2005 until 2008 he worked at IBM Austin as a verification and test engineer. His main responsibilities involved chip yield enhancement and related lab development work for the IBM Power series processors. From 2008 until present he worked at IBM Server and Technology group as a register file circuit designer. In the spring semester of 2007 he entered the Option III Graduate Program in circuit design at the University of Texas at Austin.

Permanent Address: 3501 Shoreline Drive

Austin, TX 78728

This report was typed by Tong Quan.